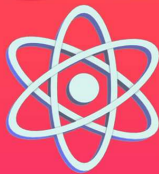


FULLSTACK REACT NATIVE



The Complete Guide to React Native



DEVIN ABBOTT HOUSSEIN DJIRDEH
ANTHONY ACCOMAZZO SOPHIA SHOEMAKER



FULLSTACK.io

Fullstack React Native

The Complete Guide to React Native

Written by Devin Abbott, Houssein Djirdeh, Anthony Accomazzo, and Sophia Shoemaker

© 2018 Fullstack.io

All rights reserved. No portion of the book manuscript may be reproduced, stored in a retrieval system, or transmitted in any form or by any means beyond the number of purchased copies, except for a single backup or archival copy. The code may be used freely in your projects, commercial or otherwise.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Published in San Francisco, California by Fullstack.io.

Contents

Book Revision	i
We'd love to hear from you!	i
Bug Reports	i
Be notified of updates via Twitter	i
Introduction	1
About This Book	2
Running Code Examples	3
Code Blocks and Context	4
Getting Help	5
Emailing Us	5
Getting Started with React Native	7
Weather App	7
Starting the project	10
Expo	11
Components	21
Custom components	41
Summary	85
React Fundamentals	87
Breaking the app into components	88
7 step process	93
Step 2: Build a static version of the app	96
Step 3: Determine what should be stateful	114
Step 4: Determine in which component each piece of state should live . .	117
Step 5: Hardcode initial states	119
Step 6: Add inverse data flow	133

CONTENTS

Updating timers	143
Deleting timers	150
Adding timing functionality	153
Add start and stop functionality	155
Methodology review	166
Core Components, Part 1	168
What are components?	168
Building an Instagram clone	169
View	178
StyleSheet	188
Text	191
TouchableOpacity	201
Image	207
ActivityIndicator	216
FlatList	222
Core Components, Part 2	237
TextInput	240
ScrollView	246
Modal	254
Core APIs, Part 1	269
Building a messaging app	269
Initializing the project	274
The app	274
Network connectivity indicator	278
The message list	293
Toolbar	320
Geolocation	333
Input Method Editor (IME)	335
Core APIs, Part 2	355
The keyboard	355
We're Done!	385
Navigation	386

CONTENTS

Navigation in React Native	386
Contact List	394
Starting the project	400
Container and Presentational components	402
Contacts	403
Profile	408
React Navigation	412
Stack navigation	413
Tab navigation	428
Drawer navigation	453
Sharing state between screens	460
Deep Linking	470
Summary	476
Animation	477
Animation challenges	477
Building a puzzle game	479
App	484
Building the Start screen	488
Building the Game screen	518
Summary	534
Gestures	535
Building the board	536
Gesture Responder System	550
PanResponder	554
Draggable component	556
Finishing the game	574
We're Done!	579
Native Modules	581
What are native modules?	581
Building a native module	584
Development environment	586
Initializing the project	587
iOS	591
Android	603

CONTENTS

JavaScript	613
Building and publishing	623
How to read this chapter	623
Building	624
Building with Expo	624
iOS	628
Android	647
Handling Updates	656
Summary	657
Appendix	658
JavaScript Versions	658
ES2015	658
ReactDOM	666
Handling Events in React Native	667
Publishing with Expo	672
Changelog	673

Book Revision

Revision 6pa

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: rn@fullstack.io¹.

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: rn@fullstack.io².

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, [follow @fullstackreact](https://twitter.com/fullstackreact)³

¹<mailto:rn@fullstack.io?Subject=React%20Native%20testimonial>

²<mailto:rn@fullstack.io?Subject=Fullstack%20React%20Native%20book%20feedback>

³<https://twitter.com/fullstackreact>

Introduction

One of the major problems that teams face when writing native mobile applications is *becoming familiar with all the different technologies*. iOS and Android - the two dominant mobile platforms - support different languages. For iOS, Apple supports the languages [Swift](https://developer.apple.com/swift/)⁴ and [Objective-C](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html)⁵. For Android, Google supports the languages [Java](https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html)⁶ and [Kotlin](https://developer.android.com/kotlin/index.html)⁷.

And the differences don't end there. These platforms have different toolchains. And they have different interfaces for the device's core functionality. Developers have to learn each platform's procedure for things like accessing the camera or checking network connectivity.

One trend is to write mobile apps that are powered by **WebViews**. These types of apps have minimal native code. Instead, the interface is a web browser running an app written in HTML, CSS, and JS. This web app can use the native wrapper to access features on the device, like the camera roll.

Tools like [Cordova](https://cordova.apache.org/)⁸ enable developers to write these hybrid apps. The advantage is that developers can write apps that run on multiple platforms. Instead of learning iOS and Android specifics, they can use HTML, CSS, and JS to write a “universal” app.

The disadvantage, though, is that it's hard to make these apps look and feel like *real* native applications. And users can tell.

While universal WebView-powered apps were built with the idea of *build once, run anywhere*, React Native was built with the goal of *learn once, write anywhere*.

React is a JavaScript framework for building rich, interactive web applications. With React Native, we can build native mobile applications for multiple platforms using

⁴<https://developer.apple.com/swift/>

⁵<https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>

⁶<https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html>

⁷<https://developer.android.com/kotlin/index.html>

⁸<https://cordova.apache.org/>

JavaScript and React. Importantly, the interfaces we build are *translated into native views*. React Native apps are not composed of WebViews.

We'll be able to share a lot of the code we write between iOS and Android. And React Native makes it easy to write code specific to each platform when the need arises. We get to use one language (JavaScript), one framework (React), one styling engine, and one toolchain to write apps for both platforms. *Learn once, write anywhere*.

At its core, React Native is composed of React components. We'll dig deep into components throughout this book, but here's an example of what a React component looks like:

```
import React from 'react';
import { StyleSheet, Text } from 'react-native';

export default class StyledText extends React.Component {
  render() {
    return (
      <Text style={styles.text}>{this.props.content}</Text>
    );
  }
}

const styles = StyleSheet.create({
  text: {
    color: 'red',
    fontWeight: 'bold',
  },
});
```

React Native **works**. It is currently being used in production at Facebook, Instagram, Microsoft, Amazon, and thousands of other companies.

About This Book

This book aims to be an extensive React Native resource. By the time you're done reading this book, you (and your team) will have everything you need to build

reliable React Native applications.

React Native is rich and feature-filled, but that also means it can be tricky to understand all of its parts. In this book, we'll walk through everything, such as installing its tools, writing components, navigating between screens, and integrating native modules.

But before we dig in, there are a few guidelines we want to give you in order to get the most out of this book. Specifically:

- how to approach the code examples
- how to get help if something goes wrong

Running Code Examples

This book comes with a library of runnable code examples. If you purchased a digital copy of this book, the code is available to download from the same place where you downloaded the book. If you purchased a physical copy, you can find download instructions right after the table of contents and before this Introduction chapter.

We use [yarn](https://yarnpkg.com/en/)⁹ to run every example in this book. This means you can type the following commands to run any example:

- `yarn start` will start the React Native packager and print a QR code. If you're on an Android mobile device, scanning this code with the [Expo](https://expo.io/)¹⁰ app will load the application. For iOS devices, see the instructions for loading apps onto your phone at the [beginning of the first chapter](#).
- `yarn run ios` will start the React Native packager and open your app in the iOS Simulator if you are using a Mac.
- `yarn run android` will start the React Native packager and open your app on a connected Android device or emulator.

In the next chapter we'll explain each of these commands in detail.

⁹<https://yarnpkg.com/en/>

¹⁰<https://expo.io/>

Code Blocks and Context

Nearly every code block in this book is pulled from a runnable code example, which you can find in the sample code. For example, here is a code block pulled from the first chapter:

weather/1/App.js

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Notice that the header of this code block states the path to the file which contains this code: code/weather/1/App.js.

This book is written with the expectation that you'll also be looking at the example code alongside the chapter. If you ever feel like you're missing the context for a code example, open up the full code file using your favorite text editor.

For example, we often need to import libraries to get our code to run. In the early chapters of the book we show these import statements, because it's not clear where the libraries are coming from otherwise. However, the later chapters of the book are more advanced and they focus on key concepts instead of repeating boilerplate code that was covered earlier in the book. If at any point you're not clear on the context, open up the code example on disk.

Getting Help

While we've made every effort to be clear, precise, and accurate you may find that when you're writing your code you run into a problem.

Generally, there are three types of problems:

- A “bug” in the book (e.g. something is explained incorrectly)
- A “bug” in our code
- A “bug” in your code

If you find an inaccuracy in our description of something, or you feel a concept isn't clear, email us! We want to make sure that the book is both accurate and clear.

Similarly, if you've found a bug in our code we definitely want to hear about it.

If you're having trouble getting your own app working (and it isn't *our* example code), this case is a bit harder for us to handle. If you're still stuck, we'd still love to hear from you, and here some tips for getting a clear, timely response.

Emailing Us

If you're emailing us asking for technical help, here's what we'd like to know:

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.8, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?

- What have you tried already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

The absolute best way to get technical support is to send us a short, self-contained example of the problem. Our preferred way to receive this would be for you to send us an [Expo Snack link](https://snack.expo.io/)¹¹. Snack is an online code editor that let's one quickly develop and demo React Native components on the browser or an actual device without having to set up a brand new project. We'll explain Expo in more detail in the next chapter.

When you've written down these things, email us at **rn@fullstack.io**. We look forward to hearing from you.

¹¹<https://snack.expo.io/>

Getting Started with React Native

Weather App

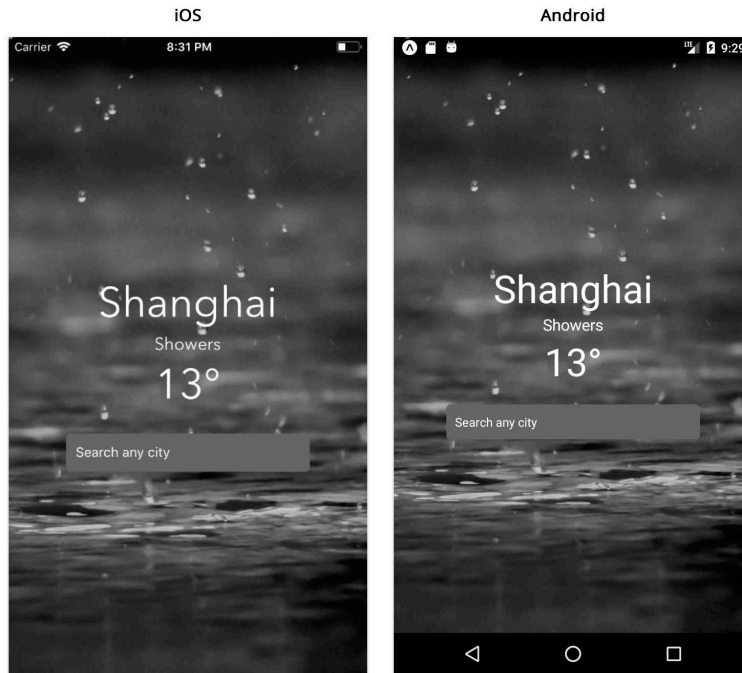
In this chapter we're going to build a weather application that allows the user to search for any city and view its current forecast.

With this simple app we'll cover some essentials of React Native including:

- Using core and custom components
- Passing data between components
- Handling component state
- Handling user input
- Applying styles to components
- Fetching data from a remote API

By the time we're finished with this chapter, you'll know how to get started by building a basic application with local state management. You'll have the foundation you need to build a wide variety of your own React Native apps.

Here's a screenshot of what our app will look like when it's done:



The completed app

In this chapter, we'll build an entire React Native application from scratch. We'll talk about how to set up our development environment and how to initialize a new React Native application. We'll also learn how Expo allows us to rapidly prototype and preview our application on our mobile device. After covering some of the basics of React Native, we'll explore how we compose apps using *components*. Components are a powerful paradigm for organizing views and managing dynamic data.

We're about to touch on a wide variety of topics, like styling and data management. This chapter will exhibit how all these topics fit together at a high-level. In subsequent chapters, we'll dive deep into the concepts that we touch on here.

Code examples

This book is example-driven. Each chapter is set up as a hands-on tutorial.

We'll be building apps from the ground up. Included with this book is a download that contains completed versions of each app as well as each of the versions we develop

along the way (the “sample code.”) If you’re following along, we recommend you use the sample code for copying and pasting longer examples or debugging unexpected errors. If you’re not following along, you can refer to the sample code for more context around a given code example.

The structure of the sample code for all the chapters in this book follows this pattern:

```
├─ components/  
├─ App.js  
├─ 1/  
|   └─ components/  
|       └─ App.js  
├─ 2/  
|   └─ components/  
|       └─ App.js  
├─ 3/  
|   └─ components/  
|       └─ App.js  
// ...
```

At the top-level of the directory is `App.js` and `components/`. This is the code for the completed version of the application. Inside the numbered folders (1/, 2/, 3/) are the different versions of the app as we build it up throughout the chapter.

Here’s what a code example in this book looks like:

weather/1/App.js

```
render() {  
  return (  
    <View style={styles.container}>  
      <Text>Open up App.js to start working on your app!</Text>  
    </View>  
  );  
}
```

Note that the title of the code block contains the path within the sample code where you can find this example (`weather/1/App.js`).

JavaScript

This book assumes some JavaScript knowledge.

React Native uses [Babel](https://babeljs.io/)¹² as a JavaScript compiler to allow us to develop in the latest version of JavaScript, regardless of what version the underlying platform supports. To understand what we mean by JavaScript versions, you can refer to the [Appendix](#).

We highlight some of JavaScript's newer features in the [Appendix](#). We reference the appendix when relevant.

Starting the project

yarn

We can install all the required tools to begin our project by using yarn. [yarn](#)¹³ is a JavaScript package manager – it *automates* the process of managing all the required dependencies from npm, an online repository of published JavaScript libraries and projects, in an application. This is done by defining all our dependencies in a single `package.json` file.

You can refer to the [documentation](#)¹⁴ for instructions to install yarn for your operating system. The documentation also explains how to install [node](#)¹⁵ as well. In order to use the Expo CLI however, Node.js v6 or later is required.

Here's a list of some commonly used yarn commands:

- `yarn init` creates a `package.json` file and adds it directly to our project.
- `yarn` installs all the dependencies listed in `package.json` into a local `node_modules` folder.
- `yarn add new-package` will install a specific package to our project as well as include it as a dependency in `package.json`. Dependencies are packages needed when we run our code.

¹²<https://babeljs.io/>

¹³<https://yarnpkg.com>

¹⁴<https://yarnpkg.com/lang/en/docs/install>

¹⁵<https://nodejs.org/en/>

- `yarn add new-package --dev` will install a specific package to our project as well as include it as a *development* dependency in `package.json`. Development dependencies are packages needed only during the development workflow. They are not needed for running our application in production.
- `yarn global add new-package` will install the package globally, rather than locally to a specific project. This is useful when we need to use a command line tool anywhere on our machine.



If you're already familiar with `npm` and have it installed, you may use it instead of `yarn` and run its [equivalent commands](#)¹⁶. These tools both manage dependencies specified in a `package.json` file. However, we find that `yarn` results in significantly more consistent builds when working with React Native, so we recommend using `yarn`.

Watchman

[Watchman](#)¹⁷ is a file watching service that watches files and triggers actions when they are modified. If you use macOS as your operating system, the Expo and React Native documentation recommend installing Watchman for better performance. The instructions to install the service can be found [here](#)¹⁸.

Expo

[Expo](#)¹⁹ is a platform that provides a number of different tools to build fully functional React Native applications without having to write native code. If you've used [Create React App](#)²⁰ before, you'll notice similarities in that no build configuration is required to get up and running.

Building an application also does not require using Xcode for iOS, or Android Studio for Android. This means that developers can build native iOS applications without

¹⁶<https://yarnpkg.com/lang/en/docs/migrating-from-npm/#toc-cli-commands-comparison>

¹⁷<https://facebook.github.io/watchman/>

¹⁸<https://facebook.github.io/watchman/docs/install.html#installing-on-os-x-via-homebrew>

¹⁹<https://expo.io/>

²⁰<https://github.com/facebookincubator/create-react-app>

even owning a Mac computer. Using Expo is the easiest way to get started with React Native and is recommended in the React Native [documentation](#)²¹.

Expo provides two local development tools that allow us to build, preview, share and publish React Native projects:

- **Expo client app:** A client iOS/Android mobile application for previewing projects.
- **Expo CLI:** A local development tool for building React Native projects with no build configuration. This is done through Create React Native App, which is merged directly with the CLI.

Including Native Code

The Expo CLI is not the only way to start a React Native application. If we need to start a project with the ability to include native code, we'll need to use the [React Native CLI](#)^a instead. With this however, our application will require Xcode and Android Studio for iOS and Android respectively.

Expo also provides a number of different APIs for device specific properties such as contacts, camera and video. However, if we need to include a native iOS or Android dependency that is not provided by Expo, we'll need to *eject* from the platform entirely. Ejecting an Expo application means we have full control of managing our native dependencies, but we would need to use the React Native CLI from that point on.

We'll explore how to use React Native CLI and add native modules to a project later on in this book.

^a<https://facebook.github.io/react-native/docs/getting-started.html#installing-dependencies>

Previewing with the Expo client

To develop and preview apps with Expo, we need to install its [client iOS or Android app](#)²² to develop and run React Native apps on our device.

²¹<https://facebook.github.io/react-native/docs/getting-started.html>

²²<https://github.com/expo/expo>

Android

On your Android mobile device, install the Expo client on [Google Play](#)²³. You can then select Scan QR code and scan this QR code once you've installed the app:



QR Code

If this QR code doesn't work, we recommend making sure you have the latest version of that Expo app installed, and that you're reading the latest edition of this book.



Instead of scanning the QR code, you can also type the project URL, `exp://exp.host/@fullstackio/weather`, inside of Expo to load the application.

iOS

You can install the Expo client via the [App Store](#)²⁴. With an iOS device however, there is no capability to scan a QR code. This means we'll first need to build the final app in order to preview it. We can do this by navigating to the `weather/` directory in the sample code folder and running the following commands:

```
cd weather
yarn
yarn start
```

This will start the React Native packager. Pressing `e` will allow you to send a link to your device by SMS or e-mail (you'll need to provide your mobile phone number

²³<https://play.google.com/store/apps/details?id=host.exp.exponent>

²⁴<https://itunes.apple.com/us/app/expo-client/id982107779?mt=8>

or email address). Once done, clicking the link will open the application in the Expo client.

For the app to load on your physical device, you'll need to make sure that your phone is connected to the same local network as your computer.

Using a supported version

The Expo client app only supports the most recent 5 or 6 releases of Expo and React Native. This book uses Expo SDK 31 (React Native version 0.57). If you're reading this book more than 6 months after the release date, you may receive an error indicating the Expo version of your project is too old if you try to run your app in the Expo client. You can upgrade to the latest version of Expo and React Native by referring to the upgrade guide for your current version in the [documentation](#)²⁵.

Most of the time, updates don't cause significant API changes. However, if you purchased a digital copy of the book, we recommend you download an updated version of the book just in case.

Preparing the app with the CLI

At this point, you should see the final application load successfully on your device. Play around with the app for a few minutes to get a feel for it. Try searching for different cities as well a location that doesn't even exist.

If you plan on building the application as you read through the chapter, you'll need to create a brand new project. Once `yarn` is installed, let's run the following command to install the Expo CLI globally:

```
yarn global add expo-cli@2.5.0
```



The `@2.5.0` specifies the *version* of `expo-cli` to install. It's important to lock in version `2.5.0` so that the version on your machine matches that here in the book.

We'll call our application `weather` and can use the following command to get started:

²⁵<https://docs.expo.io/versions/latest/workflow/upgrading-expo>

```
expo init weather --template blank@sdk-31 --yarn
```

The CLI will take a few minutes to download and extract all the project files. We used the `--template` option to specify that we want the *blank* template built with Expo SDK 31. We used the `--yarn` option to use yarn instead of `npm` for installing dependencies. Navigate to the `weather` directory once that command is finished to boot the app:

```
cd weather
yarn start
```

Once the project has finished starting, you should see some information outputted to the console.

```

▶ yarn start
yarn start v0.19.1
$ expo start
[13:15:27] Starting project at /Users/houssein/Dev/test-that-new-weather
[13:15:27] Expo DevTools is running at http://localhost:19002
[13:15:27] Opening DevTools in the browser... (press shift-d to disable)
[13:15:32] Starting Metro Bundler on port 19001.
[13:15:36] Tunnel ready.

exp://10.0.0.132:19000

```



```

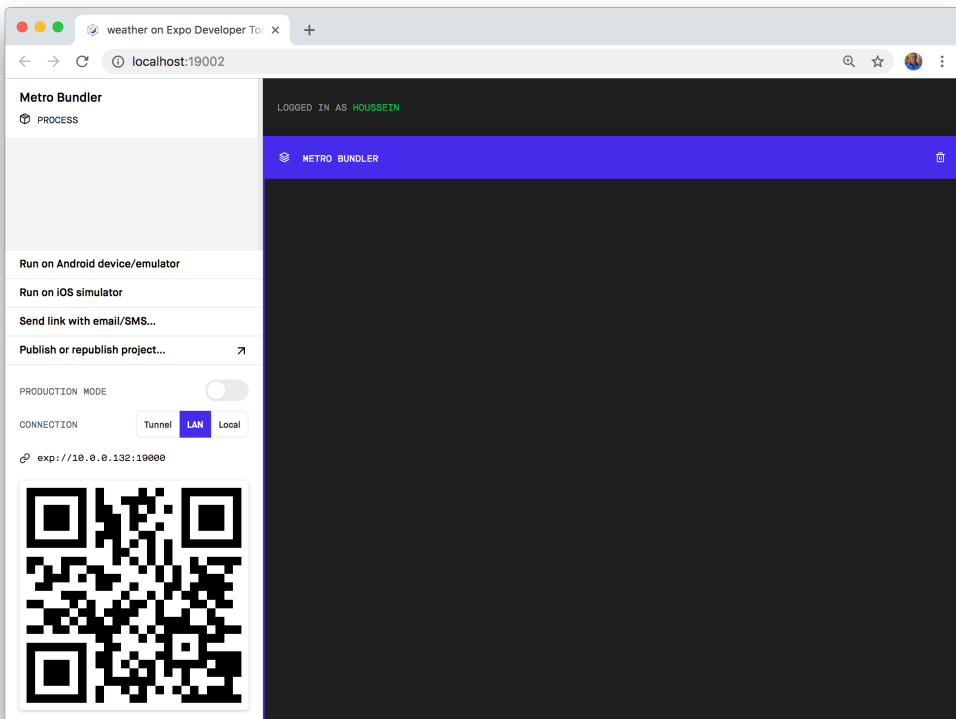
To run the app with live reloading, choose one of:
• Sign in as @houssein in Expo Client on Android or iOS. Your projects will automatically appear in the "Projects" tab.
• Scan the QR code above with the Expo app (Android) or the Camera app (iOS).
• Press a for Android emulator, or i for iOS simulator.
• Press e to send a link to your phone with email/SMS.

Press ? to show a list of all available commands.
Logs for your project will appear below. Press Ctrl+C to exit.

```

With the packager running, scanning the QR code with an Android device or sending a link directly to an iOS device using the `e` hotkey will allow you to preview the application. It is important to remember that a device needs to be connected to the same local network as the computer in order for this to work.

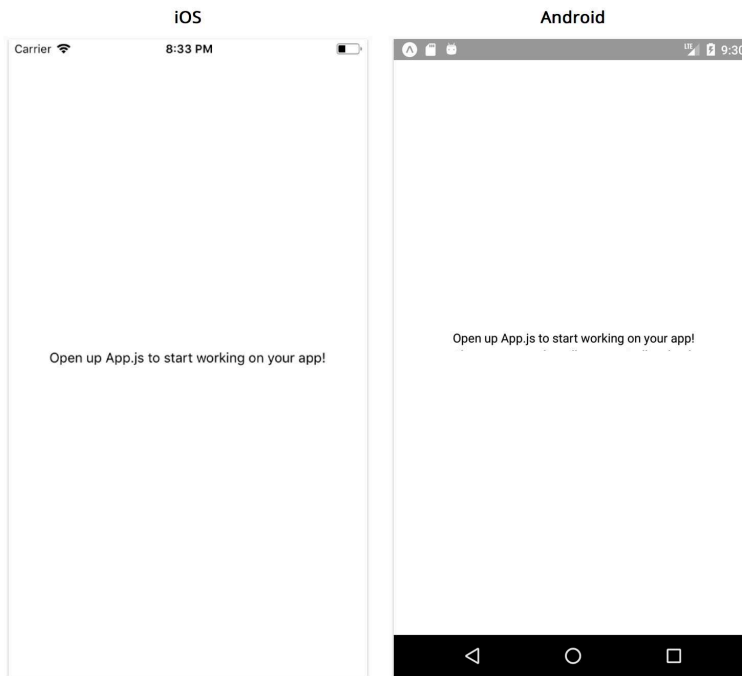
Secondly, a browser tab that renders Expo's Developer Tools should have also opened for you automatically.



Expo Developer Tools

With Expo DevTools, you can see outputted logs easier as well as perform a number of actions, such as sending an email link, directly through its interface instead of typing into the console if you prefer.

Open the application with your Android device by using the QR code or by sending a link to your iOS device. Once it finishes loading, we should see the starting point:



Application



Running on a simulator

As we mentioned, using the Expo client app allows us to run our application without using native tooling (Xcode for iOS, or Android Studio for Android).

However if we happen to have the required build tools we can still run our application in a virtual device or simulator:

- With a Mac, `yarn run ios` will start the development server and run the application in an iOS simulator. We can also start the packager separately with `yarn start` and press `i` to open the simulator.
- With the required [Android tools](#)²⁶, `yarn run android` will start the application in an Android emulator. Similarly, pressing `a` when the React Native packager is running will also boot up the emulator.

Running an application using an emulator/simulator can be useful to test on different devices and screen sizes. It can also be quicker to update and test code changes on a virtual device. However, it's important to run your application on an actual device at some point in order to get a better idea of how exactly it looks and feels.

By default, the Expo CLI comes with *live reload* enabled. This means if you edit and save any file, the application on your mobile device will **automatically reload**. Moreover, any build errors and logs will be displayed directly in the terminal.

Let's see what the directory structure of our app looks like. Open up a new terminal window.

Navigate to this app:

```
cd weather
```

And then run `ls -a` to see all the contents of the directory:

```
ls -a
```

²⁶<https://facebook.github.io/react-native/docs/getting-started.html>



If you're using PowerShell or another non-Unix shell, you can just run `ls`.

Although your output will look slightly different based on your operating system, you should see all the files in your directory listed:

```
├─ .expo/  
├─ assets/  
├─ node_modules/  
├─ .gitignore  
├─ .watchmanconfig  
├─ App.js  
├─ app.json  
├─ babel.config.js  
├─ package.json
```

Let's go through each of these files:

- `.expo/` contains files that define configurations for the Expo packager and for device connection settings.
- `.assets/` contains a few image assets by default. These are the app icon and splash screen images.
- `node_modules/` contains all third party packages in our application. Any new dependencies and development dependencies go here.
- `.gitignore` is where we specify which files should be ignored by Git. We can see that both the `node_modules/` and `.expo/` directories are already included.
- `.watchmanconfig` defines configurations for Watchman.
- `App.js` is where our application code lives.
- `app.json` is a configuration file that allows us to add information about our Expo app. The list of properties that can be included in this file is listed in the [documentation](https://docs.expo.io/versions/latest/workflow/configuration)²⁷. Examples of properties that can be changed here include the app icon and splash screen.

²⁷<https://docs.expo.io/versions/latest/workflow/configuration>

- `.babel.config.js` allows us to define presets and plugins for configuring [Babel](https://babeljs.io/)²⁸. As we mentioned previously, Babel is a transpiler that compiles newer experimental JavaScript into older versions so that it stays compatible with different platforms.
- `package.json` is where we provide information of the application to our package manager as well as specify all our project dependencies.

package.json

Let's take a closer look at the generated `package.json` file:

```
1 {
2   "name": "empty-project-template",
3   "main": "node_modules/expo/AppEntry.js",
4   "private": true,
5   "scripts": {
6     "start": "expo start",
7     "android": "expo start --android",
8     "ios": "expo start --ios",
9     "eject": "expo eject"
10  },
11  "dependencies": {
12    "expo": "^31.0.2",
13    "react": "16.5.0",
14    "react-native": "https://github.com/expo/react-native/archive/sdk-3\
15  1.0.0.tar.gz"
16  },
17  "devDependencies": {
18    "babel-preset-expo": "^5.0.0"
19  }
20 }
```

The `scripts` property contains all the commands needed to run the application. `yarn start`, `yarn run android`, and `yarn run ios` allow us to start the application

²⁸<https://babeljs.io/>

development server and/or run on a virtual device or simulator. `yarn run eject` starts the process of ejecting the application from the Expo toolchain. As we mentioned earlier, this can be necessary if we need to include a React Native library that contains native code or if we need to write native code ourselves.

The `dependencies` and `devDependencies` properties define the application and development dependencies respectively. Expo, the React core library and the version of React Native needed for this specific version of Expo are all included as dependencies. `babel-preset-expo`, a Babel preset that contains a number of predefined Babel plugins, is the only development dependency included by default.

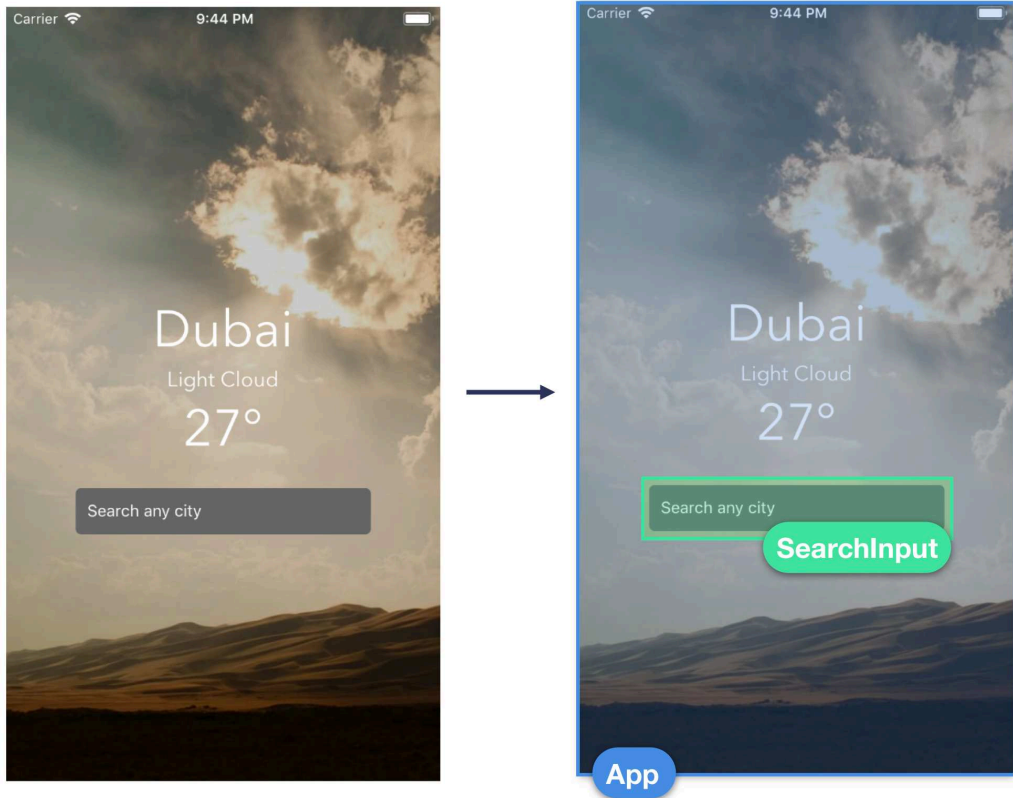
The `utils/` directory

If you look at the book's sample code, you'll note that every application has a `utils/` directory. This directory contains helper functions that the application will use. You don't need to concern yourself with the details of these functions as they're not relevant to the chapter's core concepts.

When we reach the point in the application's development where we need to use a utility provided by `utils/`, we'll remind you to copy over that folder from the sample code. You can also do this immediately after initializing each project.

Components

With newer versions of JavaScript, we can define objects with properties using *classes*. React Native lets us use this syntax to create *components*. Let's take a look at a visual breakdown of the components in our application:



Component Structure

We have an `App` component that represents the entire *screen* and contains the weather information displayed to the user. Inside of this component, we have a `SearchInput` component that allows us to search for different cities.

App

`App` is the only component created with a default application using a blank template. Let's take a look at its file:

weather/1/App.js

```
1 import React from 'react';
2 import { StyleSheet, Text, View } from 'react-native';
3
4 export default class App extends React.Component {
5   render() {
6     return (
7       <View style={styles.container}>
8         <Text>Open up App.js to start working on your app!</Text>
9       </View>
10    );
11  }
12 }
13
14 const styles = StyleSheet.create({
15   container: {
16     flex: 1,
17     backgroundColor: '#fff',
18     alignItems: 'center',
19     justifyContent: 'center',
20   },
21 });
```

Notice how we have a class defined in our file named `App` that extends `React.Component`. Using `extends` allows us to declare a class as a subclass of another class. In here, we've defined `App` as a subclass of `React.Component`. This is how we specify a specific class to be a *component* in our application.



If you'd like to learn more about how classes work in JavaScript, refer to our [Appendix](#).

We can also attach methods as properties to classes, and the same applies to component classes in React Native. We can see we already have one for this component, the `render` method:

weather/1/App.js

```
5   render() {  
6     return (  
7       <View style={styles.container}>  
8         <Text>Open up App.js to start working on your app!</Text>  
9       </View>  
10    );  
11  }
```

What we see on our device when launching our device matches what we see described in this method. **The `render()` method is the only required method for a React Native component.** React Native uses the return value from this method to determine what to render for the component.

When we use React Native, we represent different parts of our application as **components**. This means we can build our app using different reusable pieces of logic with each piece displaying a specific part of our UI. Let's break down what we already have in terms of components:

- Our entire application is rendered with `App` as our top-level component. Although created automatically as part of setting up a new Expo CLI project, this component is a custom component responsible for rendering what we need in our application.
- The `View` component is used as a layout container.
- Within `View`, we use the `Text` component to display lines of text in our application. Unlike `App`, both `View` and `Text` are **built-in** React Native components that are imported and used in our custom component.

We can see that our `App` component uses and returns an HTML-like structure. This is **JSX**, which is an extension of JavaScript that allows us to use an XML-like syntax to define our UI structure.

JSX

When we build an application with React Native, components ultimately render native views which are displayed on our device. As such, the `render()` method of a

component needs to describe how the view should be represented. In other words, React Native allows us to describe a component's iOS and Android representation in JavaScript.

JSX was created to make the JavaScript representation of components easier to understand. It allows us to structure components and show their hierarchy visually in markup. Consider this JSX snippet:

```
<View>
  <Text style={{ color: 'red' }}>
    Hello, friend! I am a basic React Native component.
  </Text>
</View>
```

In here, we've nested a `Text` component within a `View` component. Notice how we use braces (`{}`) around an object (`{ color: 'red' }`) to set the `style` property value for `Text`. In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is a JavaScript expression. The other delimiter is using quotes for strings, like this:

```
<TextInput placeholder="This is a string" />
```



Even though the JSX above might look similar to HTML, it is actually just compiled into JavaScript function calls (ex: `React.createElement(View)`). For this reason, we need to import `React` at the top of any file that contains JSX. You can refer to the [Appendix](#) for more detail.

During runtime React Native takes care of rendering the actual native UI for each component.

Props

We use the imported `Text` component to wrap each line of text output for our App component:


```
<Text>Open up App.js to start working on your app!</Text>
```

And we use the imported `View` component to wrap all the `Text` components:

```
<View style={styles.container}>  
...  
</View>
```

Props allow us to pass parameters to components to customize their features. Here, `View` is used to layout the entire content of the screen. We only have a single prop attached, `style`, that allows us to pass in style parameters to adjust how our `View` component is rendered on our devices. Each built-in component provided by React Native has its own set of valid props that we can use for customization.

If you're familiar with HTML, it's very similar. For example, in HTML, say you wanted to insert an image named `image.png`. You'd specify an `img` tag with a `src` attribute like this:

```

```

To give you an idea of the similarity, in React Native we can include images using the `Image` component. We specify the location using the `source` prop:

```
<Image source={require('./image.png')}>
```

We'll cover images in greater detail later.

Like our `View` component, many components in React Native accept a `style` prop. Styling is a large topic that we explore throughout this book. However, we can take a look at our `styles` object at the bottom of `App.js` and get an idea of how it works:

weather/1/App.js

```
14 const styles = StyleSheet.create({  
15   container: {  
16     flex: 1,  
17     backgroundColor: '#fff',  
18     alignItems: 'center',  
19     justifyContent: 'center',  
20   },  
21 });
```

Web developers may recognize that this looks like CSS (Cascading Style Sheets) which is used to style web pages. It's important to note that styling in React Native *does not* use CSS. However, React Native borrows a lot of styling nomenclature from web development. Here, we specify that the text should be centered and that the background color should be white (#fff).



If you've used CSS before, you'll find styling in React Native very familiar. If not, don't worry! It's easy to get the hang of it.



Specifically, `styles.container` has the attributes `flex`, `alignItems` and `justifyContent`. These are used to position the `View` in the center of the screen. React Native uses **flexbox** to layout and align items consistently on different device sizes. We'll go into more detail about how exactly flexbox works in later chapters.

To build our weather app, we'll start with layout and styling. Once we have some of the essence of our weather app in place we can begin to explore strategies for managing data.

As we saw in the completed version of the app, we want our app to display the **city**, **temperature**, and **weather conditions** as separate text fields. Although we'll eventually interface with a weather API in order to retrieve actual data, we'll begin with hard-coding these values.



The completed app

Adding styles

To get a better handle on styling, let's try adding an object with a color attribute to one of the text fields:

```
<View style={styles.container}>  
  <Text style={{ color: 'red' }}>  
    Open up App.js to start working on your app!  
  </Text>  
</View>
```



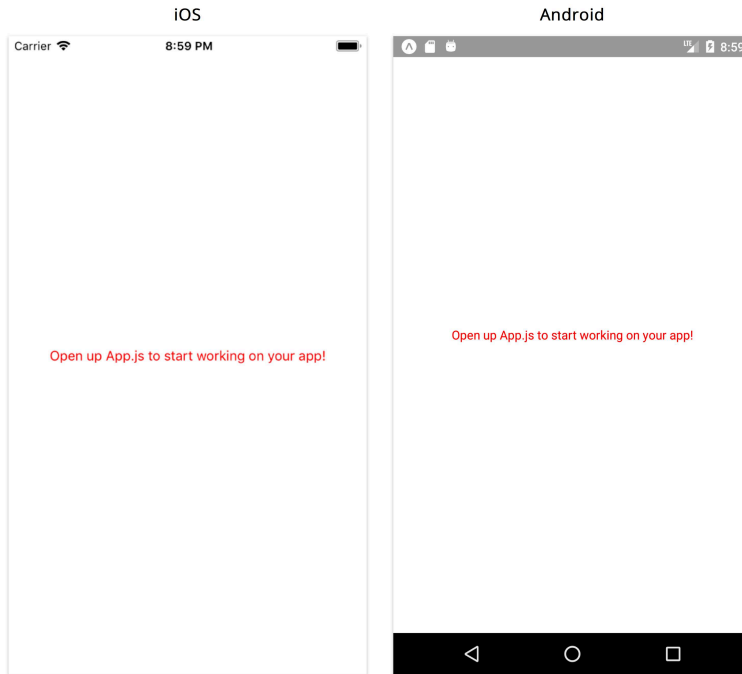
Note that the outer-most set of brackets above are *delimiters* enclosing our JavaScript statement. Inside of the delimiters is a JavaScript object. In React Native, if the object is small enough it's common to just write it all on one line.

However, the double brackets (`{{}}`) might be confusing. Here's another way of writing the same component:

```
const style = { color: 'red' };

return (
  <View style={styles.container}>
    <Text style={style}>
      Open up App.js to start working on your app!
    </Text>
  </View>
)
```

Save `App.js`. We can see our style applied once the application reloads:



As we mentioned previously, **live reload** is enabled by default in Expo. This means that with any change to the code, the application will reload immediately. If you happen to not see any changes reflected as soon as you save the file, you may have to check to see if this is enabled. The [documentation](https://docs.expo.io/versions/latest/guides/up-and-running.html#cant-see-your-changes)²⁹ explains how to open up the developer menu and enable/disable the feature.

Although we can style our entire component this way, a lot of *inline* styles (or style attributes defined directly *within* the delimiter of the `style` prop) used in a component can make things harder to read and digest.

We can solve this by leveraging React Native's `StyleSheet` API to separate our styles from our component. With `StyleSheet`, we can create styles with attributes similar to CSS stylesheets. We can see that `StyleSheet` is already imported at the top of the file. It's used to declare our first style, `styles.container`, which we use for `View`. We can add a new style called `red` to our `styles`:

²⁹<https://docs.expo.io/versions/latest/guides/up-and-running.html#cant-see-your-changes>

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  red: {
    color: 'red',
  },
});
```

We'll then have Text use this style:

```
<View style={styles.container}>
  <Text style={styles.red}>
    Open up App.js to start working on your app!
  </Text>
</View>
```

If we save our file and take a look at our app, we can see that the end result is the same.

Now let's add some appropriate styles and text fields in order to display some weather data for a location. To add multiple styles to a single component, we can pass in **an array of styles**:

weather/2/App.js

```
17     <Text style={[styles.largeText, styles.textStyle]}>
18       San Francisco
19     </Text>
20     <Text style={[styles.smallText, styles.textStyle]}>
21       Light Cloud
22     </Text>
23     <Text style={[styles.largeText, styles.textStyle]}>24°</Text>
```

It is important to mention that when passing an array, the styles at the end of the array take precedence over earlier styles, in case of any repeated attributes. We can see that we're referencing three new styles; `textStyle`, `smallText`, and `largeText`. Let's define these within our `styles` object:

weather/2/App.js

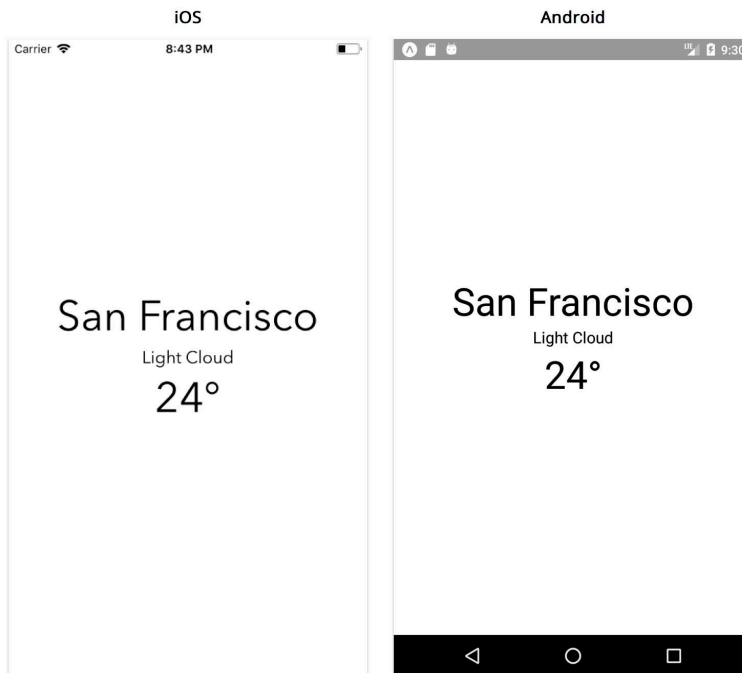
```
36 const styles = StyleSheet.create({
37   container: {
38     flex: 1,
39     backgroundColor: '#fff',
40     alignItems: 'center',
41     justifyContent: 'center',
42   },
43   textStyle: {
44     textAlign: 'center',
45     fontFamily:
46       Platform.OS === 'ios' ? 'AvenirNext-Regular' : 'Roboto',
47   },
48   largeText: {
49     fontSize: 44,
50   },
51   smallText: {
52     fontSize: 18,
53   },
```

- `TextStyle` specifies an alignment (`center`) as well as the `fontFamily`. Notice how we use `Platform` to define platform specific fonts for both iOS and Android. We do this because both operating systems provide a different set of native fonts.
- `smallText` and `largeText` both specify different font sizes.

`Platform` is a built-in React Native API. We'll need to make sure to import it:

```
import { StyleSheet, Text, View, Platform } from 'react-native';
```

Let's take a look at our application now:



Styled Text

Platform specific properties

The `Platform` API allows us to conditionally apply different styles or properties in our component based on the device's operating system. The `OS` attribute of the object returns either `ios` or `android` depending on the user's device.

Although this is a relatively simple way to apply different properties in our application based on the user's device, there may be scenarios where we may want our component to be substantially different between operating systems.

We can also use the `Platform.select` method that takes the operating system as keys within an object and returns the correct result based on the device:

```
1  textStyle: {
2    textAlign: 'center',
3    ...Platform.select({
4      ios: {
5        fontFamily: 'AvenirNext-Regular',
6      },
7      android: {
8        fontFamily: 'Roboto',
9      },
10   }),
11 },
```

Separate files

Instead of applying conditional checks using `Platform.OS` a number of times throughout the entire component file, we can also leverage the use of **platform specific files** instead. We can create two separate files to represent the same component each with a different extension: `.ios.js` and `.android.js`. If both files export the same component class name, the React Native packager knows to choose the right file based on the path extension. We'll dive deeper into platform specific differences later in this book.

Text input

We now have text fields that display the location, weather condition, and temperature. The next thing we need to do is provide some sort of input to allow the user to search for a specific city. Again, we'll continue using hardcoded data for now. We'll only begin using an API for real data once we have all of our components in place.

React Native provides a built-in `TextInput` component that we can import into our component that allows us to accept user input. Let's include it within our `View` container underneath the `Text` components (make sure to import it as well!):

weather/2/App.js

```
<Text style={[styles.largeText, styles.textStyle]}>
  San Francisco
</Text>
<Text style={[styles.smallText, styles.textStyle]}>
  Light Cloud
</Text>
<Text style={[styles.largeText, styles.textStyle]}>24°</Text>

<TextInput
  autoComplete={false}
  placeholder="Search any city"
  placeholderTextColor="white"
  style={styles.textInput}
  clearButtonMode="always"
/>
```



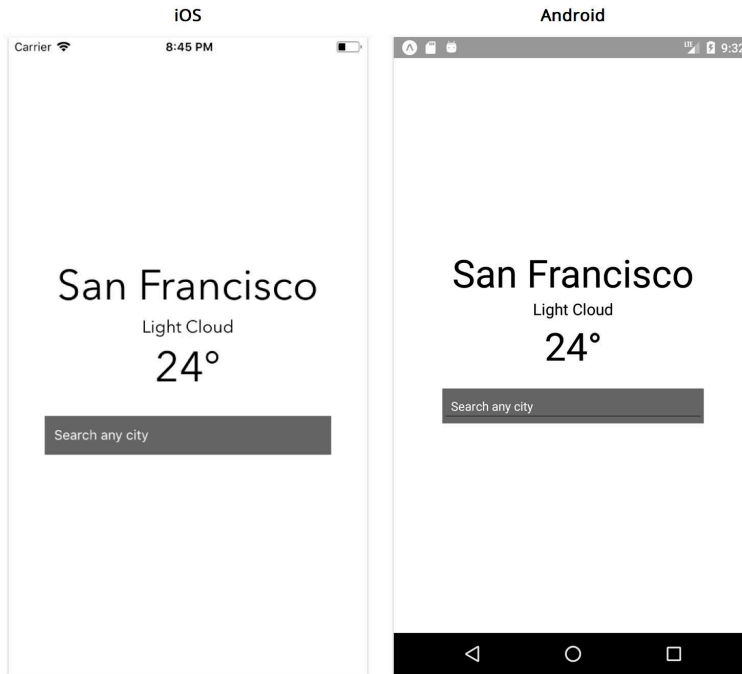
`TextInput` here is being referenced using a *self-closing* tag (`<TextInput />`). JSX allows us to use this shorthand version instead of an opening and closing tag (`<TextInput></TextInput>`) when a component has no children elements nested within.

There are a number of props associated with `TextInput` that we can use. We'll cover the basics here but go into more detail about them in the "Core Components" chapter. Here we're specifying a placeholder, its color, as well as a style for the component itself. Let's create its style object, `textInput`, underneath our other styles:

weather/2/App.js

```
smallText: {
  fontSize: 18,
},
textInput: {
  backgroundColor: '#666',
  color: 'white',
  height: 40,
  width: 300,
  marginTop: 20,
  marginHorizontal: 20,
  paddingHorizontal: 10,
  alignSelf: 'center',
},
```

As we mentioned previously, all the attributes that we provide styles with in React Native are extremely similar to how we would apply them using CSS. Now let's take a look at our application:



Text Input

We can see that the text input has a default underline on Android. We'll go over how to remove this in a bit.

We've also specified the `clearButtonMode` prop to be `always`. This shows a button on the right side of the input field when characters are inserted that allows us to clear the text. This is only available on iOS.



Text Input Clear Button

We can now type into the input field!

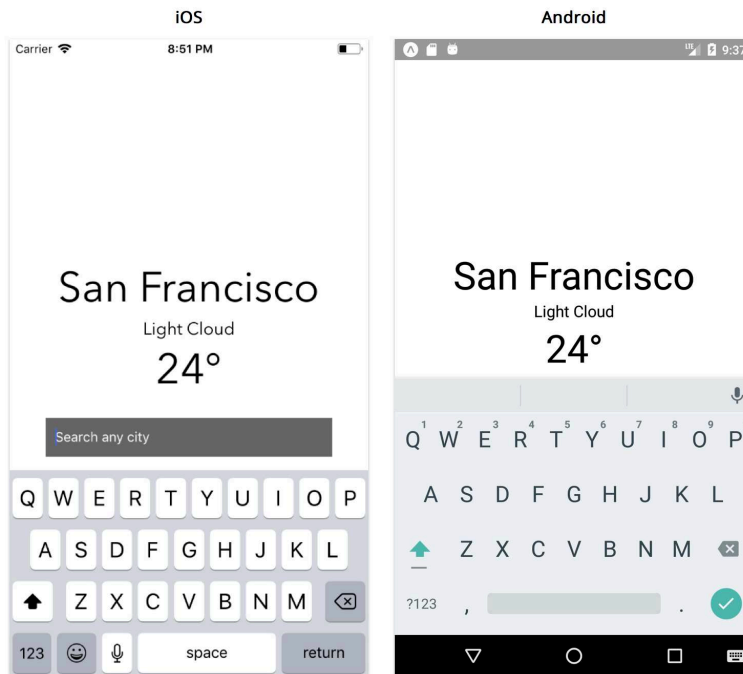


If you're using the iOS simulator, you can connect your hardware keyboard and use that with any input field. This can be done with `Shift + ⌘ + K` or going to Hardware -> Keyboard -> Connect Hardware Keyboard

With this enabled, the software keyboard may not show by default. You can toggle this by pressing `⌘ + K` or going to Hardware -> Keyboard -> Toggle Software Keyboard

Now every time you click an input field, the software keyboard will display exactly how it would if you were using a real device and you can type using your hardware keyboard.

However one thing you may have noticed is that when you focus on the input field with a tap, the keyboard pops up and covers it on Android and comes quite close on iOS:



Keyboard

Since the virtual keyboard can cover roughly half the device screen, this is a common problem that occurs when using text inputs in an application. Fortunately, React

Native includes `KeyboardAvoidingView`, a component that solves this problem by allowing us to adjust where other components render in relation to the virtual keyboard. Let's import and use this component instead of `View`:

weather/2/App.js

```
render() {  
  return (  
    <KeyboardAvoidingView  
      style={styles.container}  
      behavior="padding"  
    >  
      <Text style={[styles.largeText, styles.textStyle]}>  
        San Francisco  
      </Text>  
      <Text style={[styles.smallText, styles.textStyle]}>  
        Light Cloud  
      </Text>  
      <Text style={[styles.largeText, styles.textStyle]}>24°</Text>  
  
      <TextInput  
        autoCorrect={false}  
        placeholder="Search any city"  
        placeholderTextColor="white"  
        style={styles.textInput}  
        clearButtonMode="always"  
      />  
    </KeyboardAvoidingView>  
  );  
}
```

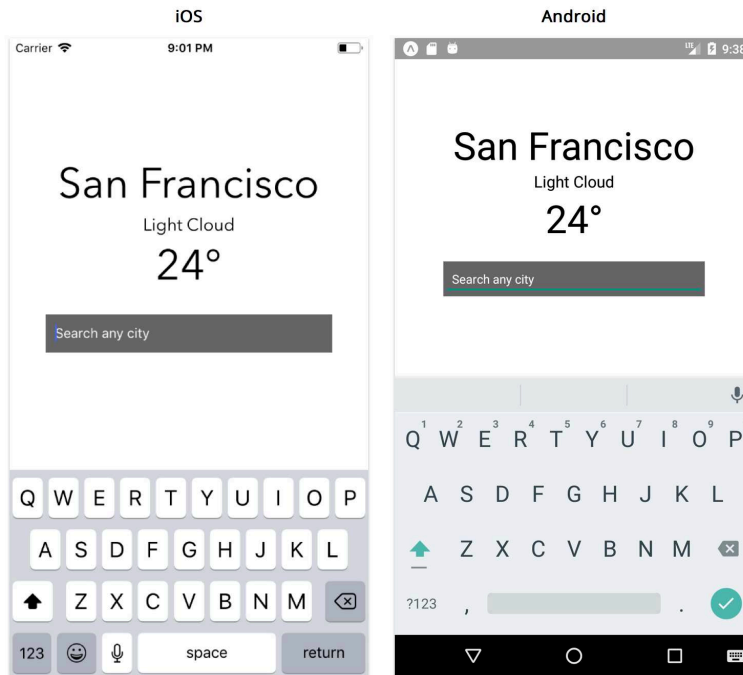
Notice that `KeyboardAvoidingView` accepts a `behavior` prop with which we can customize how the keyboard adjusts. It can change its height, position or bottom padding in relation to the position of the virtual keyboard. Here, we've specified `padding`.

And finally, it's important to double check our imports to make sure we have everything that we're using:

weather/2/App.js

```
1 import React from 'react';  
2 import {  
3   StyleSheet,  
4   Text,  
5   KeyboardAvoidingView,  
6   Platform,  
7   TextInput,  
8 } from 'react-native';
```

Now tapping the text input will shift our component text and input fields out of the way of the software keyboard.



Keyboard Avoiding View

Custom components

So far, we've explored how to add styling into our application, and we've included some built-in components into our main App component. We use `View` as our component container and import `Text` and `TextInput` components in order to display hardcoded weather data as well as an input field for the user to change locations.

It's important to re-iterate that React Native is **component-driven**. We're already representing our application in terms of components that describe different parts of our UI without too much effort, and this is because React Native provides a number of different built-in components that you can use immediately to shape and structure your application.

However, as our application begins to grow, it's important to begin thinking of how it can further be broken down into smaller and simpler chunks. We can do this by creating **custom components** that contain a small subset of our UI that we feel fits better into a separate, distinct component file. This is useful in order to allow us to further split parts of our application into something more manageable, reusable and testable.

Although our application in its current state isn't extremely large or unmanageable, there's still some room for improvement. The first way we can refactor our component is to move our `TextInput` into a separate component to hide its implementation details from the main App component. Let's create a `components` directory in the root of the application with the following file:

```
├── components/  
    └─ SearchInput.js
```

All the custom components we create that we use in our main App component will live inside this directory. For more advanced apps, we might create directories within `components` to categorize them more specifically. Since this app is pretty simple, let's use a flat `components` directory.

The `SearchInput` will be our first custom component so let's move all of our code for `TextInput` from `App.js` to `SearchInput.js`:

weather/3/components/SearchInput.js

```
1  import React from 'react';
2  import { StyleSheet, TextInput, View } from 'react-native';
3
4  export default class SearchInput extends React.Component {
5    render() {
6      return (
7        <View style={styles.container}>
8          <TextInput
9            autoComplete={false}
10             placeholder={this.props.placeholder}
11             placeholderTextColor="white"
12             underlineColorAndroid="transparent"
13             style={styles.textInput}
14             clearButtonMode="always"
15           />
16         </View>
17       );
18     }
19   }
20
21   const styles = StyleSheet.create({
22     container: {
23       height: 40,
24       width: 300,
25       marginTop: 20,
26       backgroundColor: '#666',
27       marginHorizontal: 40,
28       paddingHorizontal: 10,
29       borderRadius: 5,
30     },
31     textInput: {
32       flex: 1,
33       color: 'white',
34     },
35   });
```

Let's break down what this file contains:

- We export a component named `SearchInput`.
- This component accepts a `placeholder` prop.
- This component returns a React Native `TextInput` with a few of its properties specified wrapped within a `View`.
- We've applied the appropriate styles to our view container including a `borderRadius`.
- We also added `underlineColorAndroid="transparent"` to remove the dark underline that shows by default on Android.



this is a special keyword in JavaScript. The details about this are a bit nuanced, but for the purposes of the majority of this book, **this will be bound to the React Native component class**. So, when we write `this.props` inside the component, we're accessing the `props` property on the component. When we diverge from this rule in later sections, we'll point it out.

For more details on this, check out this page on [MDN³⁰](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this).

Custom props

As you may recall, in `App.js` we set the `placeholder` prop for `TextInput` to "Search any city." That renders the text input with a placeholder:



For `SearchInput`, we could hardcode a string again for `placeholder`. But what if we wanted to add a search input elsewhere in our application? It would be nice if `placeholder` was customizable.

³⁰<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

Earlier in this chapter, we explored how we can use props with a number of built-in components in order to customize their features. We can also *create* props for custom components that we build as well.

That's what we do here in `SearchInput`. The component accepts the prop `placeholder`. In turn, `SearchInput` uses this value to set the `placeholder` prop on `TextInput`.

The way data flows from parent to child in React Native is through props. When a parent renders a child, it can send along props the child depends on. A component can access all its props through the object `this.props`. If we decide to pass down the string "Type Here" as the `placeholder` prop, the `this.props` object will look like this:

```
{ "placeholder": "Type Here" }
```

In here, we'll set up `App` to render `SearchInput` which means that `App` is the *parent* of `SearchInput`. Our parent component will be responsible for passing down the actual value of `placeholder`.

We're getting somewhere interesting now. We've set up a custom `SearchInput` component and by building it to accept a `placeholder` prop, we're already setting it up to be configurable. Based on what it receives, it can render any placeholder message that we'd like.

Importing components

In order to use `SearchInput` in `App`, we need to import the component first. We can remove the `TextInput` logic from `App.js` and have `App` use `SearchInput` instead:

weather/3/App.js

```
import React from 'react';
import {
  StyleSheet,
  Text,
  KeyboardAvoidingView,
  Platform,
} from 'react-native';

import SearchInput from './components/SearchInput';

export default class App extends React.Component {
  render() {
    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="padding"
      >
        <Text style={[styles.largeText, styles.textStyle]}>
          San Francisco
        </Text>
        <Text style={[styles.smallText, styles.textStyle]}>
          Light Cloud
        </Text>
        <Text style={[styles.largeText, styles.textStyle]}>24°</Text>

        <SearchInput placeholder="Search any city" />
      </KeyboardAvoidingView>
    );
  }
}
```

By moving the entire `TextInput` details into a separate component called `SearchInput`, we've made sure to not have any of its specific implementation details showing in

the parent component anymore. We can also remove the text input's styling defined within the `styles` object.

There's no specific answer to how often we should isolate different UI logic into separate custom components. React Native was built in order to allow us to lay out our entire application in terms of self-contained components, and that means we should separate parts of our application into distinct units with custom functionality attached to them. This allows us to build a more manageable application that's easier to control and understand. We've isolated knowledge of our search input to the component `SearchInput` and we'll continue to isolate specific pieces of our app throughout this chapter.



It's common to separate your imports into two groups: imports from dependencies, and imports from other files in your project. That's why we put a blank line above `SearchInput`. This comes down to personal style preference.

Background image

As we saw in the photo of the completed version of the app at the beginning of this chapter, we can make our application more visually appealing by displaying a background image that represents the current weather condition.

In this book's sample code, we've included a number of images for various weather conditions. If you inspect the `weather/assets` directory, you'll find images like `clear.png`, `hail.png`, and `showers.png`.

If you're following along, copy these two folders over from the sample code into your project:

1. `weather/assets`
2. `weather/utls`



We mentioned earlier that we've included a `utls/` folder for each project in the book's sample code. This folder contains helper functions that we'll use below.



If you're on macOS or Linux, you can use `cp -r` to copy directories:

```
cp -r weather/{assets,utils} ~/react-native-projects/weather/
```

With the `assets` and `utils` folders copied over, let's update our `App` component:

`weather/4/App.js`

```
import React from 'react';
import {
  StyleSheet,
  View,
  ImageBackground,
  Text,
  KeyboardAvoidingView,
  Platform,
} from 'react-native';

import getImageForWeather from './utils/getImageForWeather';

import SearchInput from './components/SearchInput';

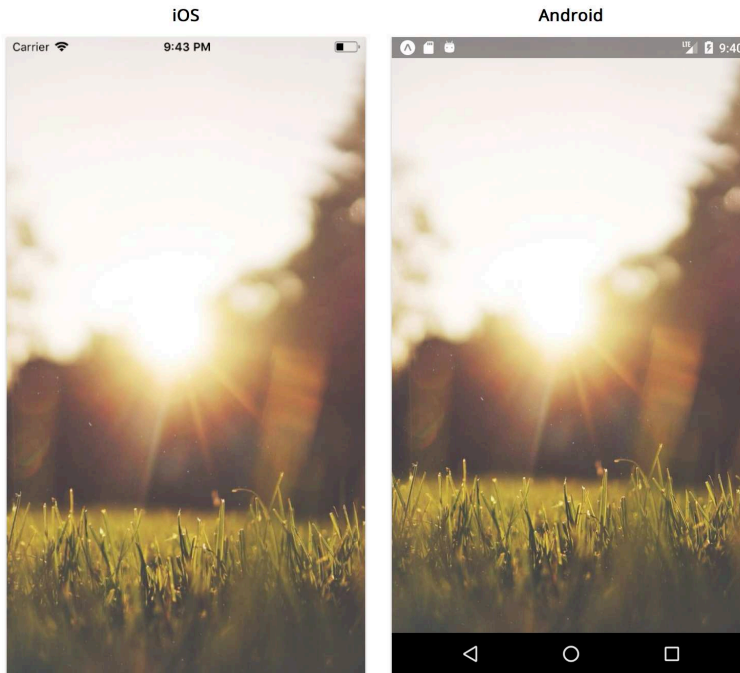
export default class App extends React.Component {
  render() {
    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="padding"
      >
        <ImageBackground
          source={getImageForWeather('Clear')}
          style={styles.imageContainer}
          imageStyle={styles.image}
        >
          <View style={styles.detailsContainer}>
```

```
    <Text style={[styles.largeText, styles.textStyle]}>
      San Francisco
    </Text>
    <Text style={[styles.smallText, styles.textStyle]}>
      Light Cloud
    </Text>
    <Text style={[styles.largeText, styles.textStyle]}>
      24°
    </Text>

    <SearchInput placeholder="Search any city" />
  </View>
</ImageBackground>
</KeyboardAvoidingView>
);
}
}
```

In this component, we're importing a `getImageForWeather` method from our `utils` directory which returns a specific image from the `assets` directory depending on a weather type.

For example, `getImageForWeather('Clear')` returns the following image:



Feel free to peek into the implementation details of any function we use from the `utils` directory to get a better idea of how it works.

We also import React Native's built-in `ImageBackground` component. Let's take a closer look at how we're making use of it in our render method:

`weather/4/App.js`

```
render() {  
  return (  
    <KeyboardAvoidingView  
      style={styles.container}  
      behavior="padding"  
    >  
      <ImageBackground  
        source={getImageForWeather('Clear')}  
        style={styles.imageContainer}
```



```
      imageStyle={styles.image}  
    >
```

Conceptually, the `ImageBackground` component is a `View` with an `Image` nested within.

The `source` prop accepts an image location, which we've set to:

```
getImageForWeather('Clear') .
```

We know this will always return the image displayed above. `ImageBackground` also uses the prop `style` for styling the `View` container and the prop `imageStyle` for styling the image itself. Let's add two new styles and modify the container style:

`weather/4/App.js`

```
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
    backgroundColor: '#34495E',  
  },  
  imageContainer: {  
    flex: 1,  
  },  
  image: {  
    flex: 1,  
    width: null,  
    height: null,  
    resizeMode: 'cover',  
  },  
},
```

Defining component styles with a `flex` attribute mean that they will expand to take up any room remaining in their parent container in relation to any sibling components. They share this space in proportion to their defined `flex` values. Since `ImageBackground` is the only nested element within `KeyboardAvoidingView`, setting `imageContainer` to `flex: 1` means that this element will fill up the entire space of its parent component. We've removed `justifyContent` and `alignItems` from `container` so that the `ImageBackground` can take up the entire device screen.

We also used `flex: 1` to style the actual image itself, `image`, to make sure it takes up the entire space of its parent container. With images in particular, the component will fetch and use the actual width and height of the source image by default. For this reason, we've also set its `height` and `width` attributes to `null` so that the dimensions of the image fit the container instead. The `resizeMode` attribute allows us to define how the image is resized when the `Image` element does not match its actual dimensions. Setting this attribute to `cover` means that the image will scale uniformly until it is equal to the size of the component.



The “Core Components” chapter will dive deeper into how flexbox, layout, and the `Image` component work in React Native

We also wrapped all of our `Text` elements and `SearchInput` within a view container styled with `detailsContainer`:

`weather/4/App.js`

```
<View style={styles.detailsContainer}>
  <Text style={[styles.largeText, styles.textStyle]}>
    San Francisco
  </Text>
  <Text style={[styles.smallText, styles.textStyle]}>
    Light Cloud
  </Text>
  <Text style={[styles.largeText, styles.textStyle]}>
    24°
  </Text>

  <SearchInput placeholder="Search any city" />
</View>
```

Now let's set up its style:

weather/4/App.js

```
detailsContainer: {  
  flex: 1,  
  justifyContent: 'center',  
  backgroundColor: 'rgba(0,0,0,0.2)',  
  paddingHorizontal: 20,  
},
```

Here, we're ensuring the container within `ImageBackground` also fills up the entire space of its parent component as well as have its items aligned at the center of the screen. We also add a semi-transparent overlay to our image by setting the `backgroundColor` of this component.

The last thing we'll need to do here is change our `Text` elements to white instead of black to show more clearly with a background image:

weather/4/App.js

```
textStyle: {  
  textAlign: 'center',  
  fontFamily:  
    Platform.OS === 'ios' ? 'AvenirNext-Regular' : 'Roboto',  
  color: 'white',  
},
```

Try it out

Save the file and take a look at our app. We should now see the background image displayed!

Modifying location

The steps we've taken so far are quite common when starting React Native applications. We hardcode all our data, organize our app into components, and get an idea of the visual layout as well as how it breaks down into components.

However, our app really isn't very useful at this moment. If we take a look at our `SearchInput` component for instance, we can type anything into the input field but nothing actually happens as a result. We need to find a way to track changes made to the component and store that information somewhere. In other words, we need some piece of **mutable data** that updates whenever the user changes or submits the input field.

Instead of having `SearchInput` not actually manage any data that represents the text inputted by the user, let's pass in a prop for it called `location` to reflect what the user has inputted into the text input field:

```
render() {  
  const location = 'San Francisco';  
  
  return (  
    <KeyboardAvoidingView  
      style={styles.container}  
      behavior="padding"  
    >  
      <ImageBackground  
        source={getImageForWeather('Clear')}  
        style={styles.imageContainer}  
        imageStyle={styles.image}  
      >  
        <View style={styles.detailsContainer}>  
          <Text style={[styles.largeText, styles.textStyle]}>  
            {location}  
          </Text>  
          <Text style={[styles.smallText, styles.textStyle]}>  
            Light Cloud  
          </Text>  
          <Text style={[styles.largeText, styles.textStyle]}>  
            24°  
          </Text>  
  
          <SearchInput placeholder="Search any city" />  
        </View>  
      </ImageBackground>  
    </KeyboardAvoidingView>  
  );  
}
```

```
    </ImageBackground>
  </KeyboardAvoidingView>
```

The reason we want to pass in the property that contains our location data is we need a way for our child component to modify that field and communicate back up to our container App component. Notice how we've moved the static string for location into a separate constant which we pass down to SearchInput. We've instantiated it as San Francisco so that it can show as the first location when the user loads the application. The next thing we just need to do is make sure that this location constant is updated when the user actually changes the field in SearchInput:

```
export default class SearchInput extends React.Component {
  handleChangeText(newLocation) {
    // We need to do something with newLocation
  }

  render() {
    return (
      <TextInput
        autoComplete={false}
        placeholder={this.props.placeholder}
        placeholderTextColor="white"
        underlineColorAndroid="transparent"
        style={styles.textInput}
        clearButtonMode="always"
        onChangeText={this.handleChangeText}
      />
    );
  }
}
```

So what did we just do? We've just added `onChangeText` as a new prop to our `TextInput` component. Notice that we don't pass in a specific object or property, but a *function* instead:

```
onChangeText={this.handleChangeText}
```

This method is invoked everytime the text within the input field is changed. A number of built-in components provided by React Native include *event-driven* props which we can attach specific methods to. We'll explore more throughout this book.

With `onChangeText`, our `TextInput` returns the changed text as an argument which we're attempting to pass into a separate method called `handleChangeText`. Currently our method is blank and we'll explore how we can complete it in a bit.

In React Native, we need to pass in functions when we want to handle certain events related to the component being referenced. For the `TextInput` component, `onChangeText` is set to fire every single time the text within the input field has changed. We need to "listen" to this specific event in our child component (`TextInput`) so that it can notify our parent component (`SearchInput`) to respond to this event. To do this, we pass in a function that calls another function, or in other words, a callback.

This is a common pattern when building components which need to notify a parent component of some event. Unfortunately with the way we've just set it up, it wouldn't work in this example. This is because the function `handleChangeText` has a different local scope than the component instance. We can work around this by binding our function to the correct context of its `this` object.

```
<TextInput
  placeholder={placeholder}
  placeholderTextColor="white"
  underlineColorAndroid="transparent"
  style={styles.textInput}
  clearButtonMode="always"
  onChangeText={this.handleChangeText.bind(this)}
/>
```

Now this might seem okay for the current context, but it can quickly become unwieldy if we build our components with `bind` statements in each event handler. One reason why is if we wanted to use `handleChangeText` in multiple different sub-components for example, we would have to make sure to bind it to the correct context

every single time. To help solve this, we can take care of handling our event using **property initializers**:

```
export default class SearchInput extends React.Component {  
  handleChangeText = (newLocation) => {  
    // We need to do something with newLocation  
  }  
  
  render() {  
    return (  
      <TextInput  
        autoCorrect={false}  
        placeholder={this.props.placeholder}  
        placeholderTextColor="white"  
        underlineColorAndroid="transparent"  
        style={styles.textInput}  
        clearButtonMode="always"  
        onChangeText={this.handleChangeText}  
      />  
    );  
  }  
}
```

This allows us to declare the member methods as arrow functions:

```
handleChangeText = (newLocation) => {  
  // We need to do something with newLocation  
}
```

And we pass the method name to the prop and nothing more:

```
onChangeText={this.handleChangeText}
```



Property Initializers

Supported by [Babel³¹](#), property initializers are still in the proposal phase and have not yet been slated for adoption in future JavaScript versions. Although this pattern is used quite often in many React and React Native applications, it is important to keep in mind that it is still *experimental* syntax.

For more information on the different ways to handle events in React Native, refer to the [Appendix](#).

Now that we've set up our callback correctly, let's modify `handleChangeText` to change our `text` prop in order to change the data to match what the user is typing:

```
handleChangeText = (newLocation) => {  
  this.props.location = newLocation;  
};
```

Let's run our application and try typing into the `TextInput` field. You'll immediately notice that the first location that shows is San Francisco, so we know that the `text` prop is being passed down successfully!

However, if we type anything into our `TextInput`, you'll notice *nothing happens*. Changing the text within the input field does not actually update the parent `location` property and from the way we've designed our component logic, it looks like it should. This is because `this.props`, which is referenced in `SearchInput`, **is actually owned by App and not the child component, SearchInput**. A component's props are **immutable** and create a one-way data pipeline from parent to children.

We have a bit of a problem. We need to find a way to:

- store local data in our child component, `SearchInput`, that represents the value in the input field
- track changes to the search input field as it's updated by the user
- notify our parent component, `App`, whenever our location changes

This is where we can use a component's **state**.

³¹<https://babeljs.io/docs/plugins/transform-class-properties/>

Storing local data

Let's modify our `SearchInput` component once more. Currently the text input within the component does nothing, so let's add some local component state to control actual data. We can do this by adding a constructor method to the component. We can then initialize the component's state within this method:

`weather/5/components/SearchInput.js`

```
export default class SearchInput extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      text: '',  
    };  
  }  
}
```

We can use the `constructor` method to initialize our **component-specific data**, or state. We do this here because this method fires before our component is mounted and rendered. Here, we defined our state object to only contain a `text` property:



Remember, components in React Native are extended from `React.Component` to create derived classes. `super()` is required in derived classes in order to reference `this` within the constructor.

Much like how we can access the component's props with `this.props`, we can access the component's state via `this.state`. For example if we wanted to output our state property in a single `Text` component, we could do this:

```

export default class HiThere extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      text: 'Hi there!',
    };
  }

  render() {
    return <Text>{this.state.text}</Text>;
  }
}

```

This component would now render 'Hi there!' since that's how we defined our `state.text` property in our constructor. For our current component however, our `text` property in state will be used to define the text typed by the user into the input field. Let's now modify our component's `render` method to allow for this:

weather/5/components/SearchInput.js

```

render() {
  const { placeholder } = this.props;
  const { text } = this.state;

  return (
    <View style={styles.container}>
      <TextInput
        autoCorrect={false}
        value={text}
        placeholder={placeholder}
        placeholderTextColor="white"
        underlineColorAndroid="transparent"
        style={styles.textInput}
        clearButtonMode="always"
        onChangeText={this.handleChangeText}
        onSubmitEditing={this.handleSubmitEditing}
      />
    </View>
  );
}

```

```
    </View>
  );
}
```

The first thing we did was destructure the component props and state objects:

weather/5/components/SearchInput.js

```
render() {
  const { placeholder } = this.props;
  const { text } = this.state;
```



Destructuring

Instead of using `this.props.placeholder` and `this.state.text` directly, we destructured both objects at the beginning of our `render` method into individual variables (`text` and `placeholder`). Please refer to the [Appendix](#) for more details on destructuring assignments.

We then make sure that the `TextInput` placeholder prop is still accepting our `props.placeholder` attribute. We also pass `state.text` to a value prop:

weather/5/components/SearchInput.js

```
<TextInput
  autoComplete={false}
  value={text}
  placeholder={placeholder}
  placeholderTextColor="white"
  underlineColorAndroid="transparent"
  style={styles.textInput}
  clearButtonMode="always"
  onChangeText={this.handleChangeText}
  onSubmitEditing={this.handleSubmitEditing}
/>
```

The `value` prop is responsible for the content showed in the input field. With this, we now know whatever is displayed in input field will **always represent our local state**.

We've also attached two additional props to our component, `onChangeText` and `onSubmitEditing` with methods we haven't set up yet.

Tracking changes to input

Let's take a look at how `onChangeText` can allow us to update our state every time the input field is changed. As we just did previously, we're attaching a method to the `onChangeText` prop of `TextInput`:

weather/5/components/SearchInput.js

```
onChangeText={this.handleChangeText}
```

Previously, we set up a `handleChangeText` method that modifies our location prop value when the user changes the text within the input. We quickly realized that this didn't work. This is because props are immutable and are always **“owned” by a component's parent** while state can be mutated and is **“owned” by the component itself**. This is an extremely important pattern to remember while building components with React Native.

This brings us to `setState()`, a method we can use to **change our state** correctly. Let's make use of this in our `handleChangeText` method which we can declare right underneath our constructor:

weather/5/components/SearchInput.js

```
export default class SearchInput extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      text: '',  
    };  
  }  
}
```

```
handleChangeText = text => {  
  this.setState({ text });  
};
```



Shorthand property names

With later versions of JavaScript, we can define objects using shorthand form where possible. Our `handleChangeText` method can also be written in a more explicit syntax:

```
handleChangeText = (text) => {  
  this.setState({ text: text });  
};
```

Please refer to the [Appendix](#) for a little more detail on this concept.

Now we might be tempted to update our state by using `this.state.text = text`, but this will **not work**. For all state modifications after the initial state we've defined in our constructor, React provides components with the method `setState()` to do this. In addition to mutating the component's state object, this method triggers the React component to re-render, which is essential after the state changes.

It's good practice to initialize components with “empty” state as we've done in this component. However, after our `SearchInput` component is initialized, we want to update the state with data the user types into the text input. This is why we use the `text` argument provided into our callback method as part of the `onChangeText` prop and pass that into `this.setState()`.



Never modify state outside of `this.setState()`. This function has important hooks around state modification that we would be bypassing.

We discuss state management in detail throughout the book.

Notifying the parent component

So we've found a way to correctly store local state in our component that represents the text within the search input *and* make sure that it updates as the user changes the

value. We still need to do one more thing which is to notify our parent App component when the user submits a new searched value. This is why we've attached a method to the `onSubmitEditing` prop of `TextInput`:

weather/5/components/SearchInput.js

```
onSubmitEditing={this.handleSubmitEditing}
```

The idea here is we don't necessarily want to communicate with our parent component everytime the user changes the input field. That's why `onChangeText` is purely responsible for storing the latest typed input value into the local state of the component. Fortunately, the `TextInput` component has an `onSubmitEditing` prop which fires when the user **submits** the field and not just changes it. This happens specifically when the user presses the action button of the virtual keyboard in order to *submit* their input. This is where we would want to notify our container component of the typed user data. Let's take a look at how we can set up the `handleSubmitEditing` function that we're passing in:

weather/5/components/SearchInput.js

```
handleSubmitEditing = () => {  
  const { onSubmit } = this.props;  
  const { text } = this.state;  
  
  if (!text) return;  
  
  onSubmit(text);  
  this.setState({ text: '' });  
};
```

In here, we check if `this.state.text` is not blank (which means the user has typed something into the field), and if that's the case:

1. **Run an `onSubmit` function obtained from the component's props.** We pass `text` as an argument here.
2. Clear the `text` property in state using `this.setState()`

We've seen how `this.props` can be used to pass information down from a parent component to child and we've also seen how built-in components such as `TextInput` can notify their parent component through callbacks in some of their props. Similarly, we can create props in our custom components to do the *exact same thing*. In here, we need `SearchInput` to communicate with the `App` component whenever the user submits the input field. We do this because we want our parent component to handle the event of the user typing and submitting a new city. This is why we have an `onSubmit` prop here that gets fired.

The next thing we need to do is pass a method to the `onSubmit` prop of `SearchInput` in `App` and handle the event:

weather/5/App.js

```
<SearchInput
  placeholder="Search any city"
  onSubmit={this.handleUpdateLocation}
/>
```

Let's define local state for this component as well as the `handleUpdateLocation` method:

weather/5/App.js

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: 'San Francisco',
    };
  }

  handleUpdateLocation = city => {
    this.setState({
      location: city,
    });
  };
};
```

We defined local state for this component with just a `location` property and have it set to San Francisco. We do this to ensure that an initial location is shown when we reload our application. We also included a `handleUpdateLocation` method that takes in a parameter to change our location state. This method will fire everytime the user submits the search input field because we pass this method as the `onSubmit` prop for `SearchInput`.

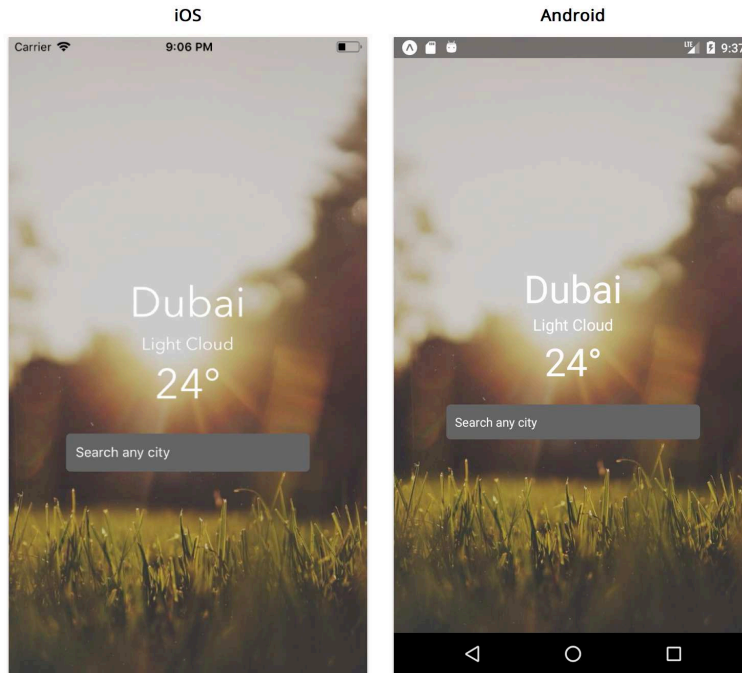
Since we actually have “living” location data represented by what the user submits in the input field, we can now display it in our first `Text` element instead of a hardcoded string:

weather/5/App.js

```
render() {  
  const { location } = this.state;  
  
  return (  
    <KeyboardAvoidingView  
      style={styles.container}  
      behavior="padding"  
    >  
      <ImageBackground  
        source={getImageForWeather('Clear')}  
        style={styles.imageContainer}  
        imageStyle={styles.image}  
      >  
        <View style={styles.detailsContainer}>  
          <Text style={[styles.largeText, styles.textStyle]}>
```

Try it out

If we type any city in the search input and press return, we'll see the name of the city being displayed immediately.



Component State

This shows that we've wired everything correctly!

We've sequenced each of our problems step by step and showed how props and state differ when trying to pass and store component data. However, `handleUpdateLocation` doesn't really get *real* weather information and just updates the city name that's being displayed. We'll wire it up to get actual weather data soon.

Architecting state

We may have already considered controlling all the location state within `SearchInput` and not having to deal with passing information upwards to a container component. There's no specific answer to *where each piece of state should live* and it depends on the type of application we're building. This is a core concept of building React Native applications and tools like [Redux](https://github.com/reactjs/redux)³² and [MobX](https://github.com/mobxjs/mobx)³³ aim to simplify this even further by allowing you to manage the entire state of the application in a single

³²<https://github.com/reactjs/redux>

³³<https://github.com/mobxjs/mobx>

GET THE FULL BOOK

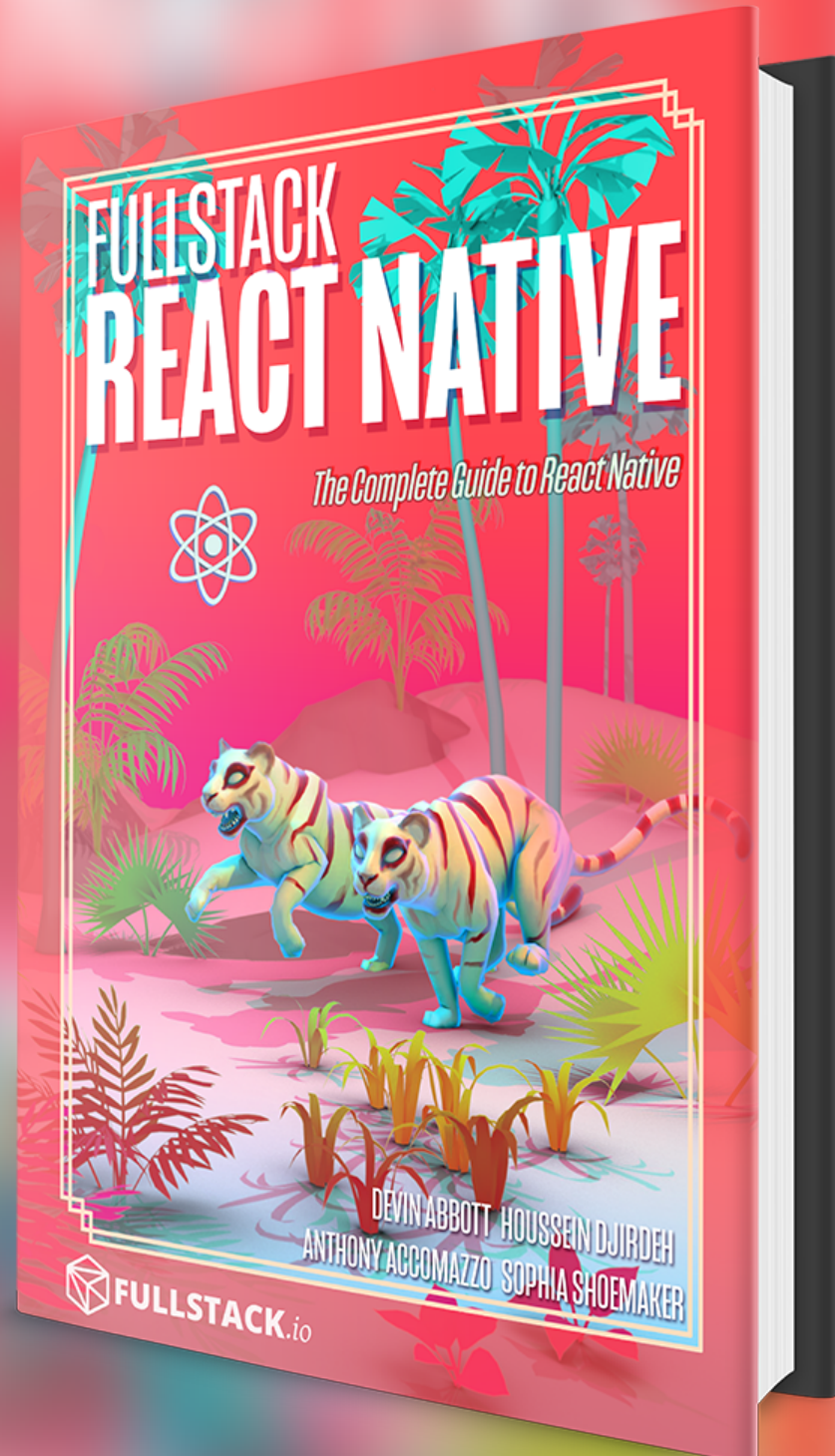
This is the end of the preview chapter!

Head over to:

<https://fullstackreact.com/react-native>
to download the full package!

Learn how to:

- * Use core device functionality such as the camera, GPS, and keyboards
- * Render rich views with dynamic data
- * Use React Native's powerful styling functionality to create gorgeous interfaces
- * Manage pages with Navigation, which is like a router for mobile apps
- * Create snappy interactions with animations
- * Integrate with platform native libraries and code
- * ... and much more!



GET IT NOW